

CM20167

Topic 11: Types in the λ -calculus

Guy McCusker

1W2.1

Types

So far, everything we've looked at has been *untyped*:

- ▶ in the λ -calculus, every term is regarded as being of the same kind, and we can apply anything as a function to anything else
- ▶ in Lisp, although there are distinctions between symbols, numbers, lists and functions, there was no *static* type checking.

Many programming languages are not like this: often the data we manipulate, and the things that manipulate it, come equipped with *types*.

Types in Java

In Java, for instance, every piece of data has (at least one) type:

- ▶ there are basic types for basic data: `int`, `boolean`, `char` etc
- ▶ names of classes are types for the objects of those classes: `String`, `ArrayList`, `MyHomeMadeClass` etc.
- ▶ the type system is quite sophisticated, including things like interfaces, subclassing, generics and so on.

Static type checking

One advantage of a type mechanism like Java's is that *the compiler can perform some checks on your code* without running the code.

This means that some programming mistakes can be caught at compile-time, without ever executing the code.

For example, Java's type system makes it quite hard to make an old C blunder:

```
if (x = 0) { // should have used ==  
  
    System.out.println("Not this old chestnut again!");  
  
}
```

Dynamic type checking

Languages like Lisp and Scheme *do* have a notion of type—functions and numbers are not the same thing, for instance—but they do not perform any compile-time type checking.

```
user> (defun badidea (x) (x (+ x 2)))  
badidea
```

The interpreter is happy with this definition of a function even though it is fatally flawed: it tries to use x both as a function and as a number, but x will never be both. We get a type error when we try to use this function.

```
user> (badidea 3)  
Continuable error---calling default handler:  
Condition class is #<class bad-type>  
message:      "incorrect type in function application"  
value:        3  
expected-type: #<class function>
```

Typed λ -calculus

We can add the idea of static type-checking to the λ -calculus relatively easily.

Indeed, the λ -calculus and variants of it have been the primary vehicle for study of types, type systems and other ways of constraining and reasoning about programs over the last 40 years.

There are a vast number of different kinds of typed λ -calculus. We will look at the simplest, which is called the *simply typed λ -calculus*.

Types for λ -terms

What should the types that we give to λ -terms look like?

A typical λ -term is a function, which will receive a term as its input and provide a term as its output.

If types are supposed to give us some kind of information about a term, then the typical information we might want of a function is:

- ▶ what type of input does it need?
- ▶ what type of output does it give?

Types for λ -terms

Thus a function-term will get a type that looks like $\alpha \rightarrow \beta$ where

- ▶ α is another type, which is the type of the input for the function
- ▶ β is another type, which is the type of the output for the function.

There might be a range of other (non-function) data that our terms manipulate, and we can give those things types too. Let's just say that there's only one type which is not a function-type, and call this type 0.

Types for λ -terms

We can now give an inductive definition of the types of λ -terms:

$$\alpha := 0 \mid \alpha \rightarrow \alpha.$$

This just says that

- ▶ 0 is a type
- ▶ if α_1 and α_2 are types, then $\alpha_1 \rightarrow \alpha_2$ are types
- ▶ only things you can build in this way are types.

Types for λ -terms

As for terms, we use parentheses to make clear the structure of a type.

Examples

- ▶ $0 \rightarrow 0$ is a type
- ▶ $0 \rightarrow (0 \rightarrow 0)$ is a type
- ▶ $(0 \rightarrow 0) \rightarrow (0 \rightarrow 0)$ is a type

We also employ a convention to reduce the number of parentheses needed: the \rightarrow operation associates *right* so that

$$0 \rightarrow 0 \rightarrow 0$$

means the same as

$$0 \rightarrow (0 \rightarrow 0).$$

Typed terms

Now that we have our collection of types, let's see how to build typed λ -terms.

In the untyped case, we used the symbol Λ for the set of untyped λ -terms.

For each type we will now define a set Λ_α of *typed λ -terms of type α* .

In the λ -calculus, the interesting work of transporting information around is done by the variables. Therefore, in the typed case, we need to record what type of term a variable can be bound to.

We therefore begin by setting up a collection of variables x^α which can be bound to terms of type α . We need an infinite number of these for each type.

Typed terms

The sets Λ_α are defined as follows:

- ▶ a variable x^α is a member of Λ_α
- ▶ if a term M is a member of Λ_β then the term $\lambda x^\alpha.M$ is a member of $\Lambda_{\alpha \rightarrow \beta}$
- ▶ if $M \in \Lambda_{\alpha \rightarrow \beta}$ and $N \in \Lambda_\alpha$ then $MN \in \Lambda_\beta$.

If $M \in \Lambda_\alpha$ we say that M is a *well-typed term with type α* .

Note that a typed λ -term is just like an ordinary, untyped λ -term except that the variables are annotated with type information.

It is a fact that if we start with an untyped term and annotate all its variables with type information, the resulting term has at most one type; that is, if it is well-typed at all, its type is unique.

Typed terms

Examples

- ▶ For any type α , the term $\lambda x^\alpha. x^\alpha$ is a term of type $\alpha \rightarrow \alpha$
- ▶ The term $\lambda x^\alpha. \lambda y^\beta. x^\alpha$ is a term of type $\alpha \rightarrow \beta \rightarrow \alpha$.
- ▶ The term $\lambda x^{(\alpha \rightarrow \beta \rightarrow \gamma)}. \lambda y^{\alpha \rightarrow \beta}. \lambda z^\alpha. x^{\alpha \rightarrow \beta \rightarrow \gamma} z^\alpha (y^{\alpha \rightarrow \beta} z^\alpha)$ is a term of type

$$(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma.$$

A remark on association

Note that application associates *left*, so MNP means $(MN)P$, while the type former \rightarrow associates *right*, so that $\alpha \rightarrow \beta \rightarrow \gamma$ means $\alpha \rightarrow (\beta \rightarrow \gamma)$.

This is perfectly natural! Consider typing the term $(MN)P$.

For this to be well-typed with type γ , we need P to have some type β , and (MN) to have type $\beta \rightarrow \gamma$.

For this to be the case we need N to have some type α , and then M must have type $\alpha \rightarrow (\beta \rightarrow \gamma)$.

Thus the natural association of application to the left corresponds to a natural association of \rightarrow to the right.

Making the notation easier

To ease the notation, we often leave off some of the type information from a term.

For instance, it gets very irritating to annotate every variable, so we might reasonably not bother to write the superscripts on every use of every variable.

More interestingly, we might not bother to annotate a variable with a type at all. But then if we just write something like

$$\lambda x. x$$

we've underspecified the term: x could be a variable of any type α , and this would then be a valid term of type $\alpha \rightarrow \alpha$.

Omitted type information. . .

This leads to interesting questions:

- ▶ Can we compute enough type information from the ways in which these un-annotated terms are used? This is the *type inference* or *type reconstruction* problem. Most real typed functional programming systems do some of this work, so that the programmer doesn't have to key in lots of boring type information.
- ▶ Can we allow the same underspecified term to be used with more than one type? This leads to *polymorphism* which comes in various guises.

Theory of typed λ -calculus

We can now develop the theory of the typed λ -calculus in the same way as we did for untyped λ -calculus:

- ▶ we define α -conversion as before; of course when we convert a term $\lambda x^\beta.M$ we have to rename x^β with another variable of the same type β
- ▶ we define β -reduction as before: note that in a typed term $(\lambda x^\alpha.M)N$, it must be the case that N has type α so we only need substitution $M[N/x^\alpha]$ in the case when the term being "plugged in" has the right type
- ▶ we get very annoyed that α and β are used both for the names of types and for the names of the key operations of the λ -calculus.

There's not much to it!

That is pretty much all we need to define the typed λ -calculus.

β -reduction just goes along as in the untyped calculus: we can show that if M is a well-typed term of type α , and it contains a redex, then reducing that redex to reach M' gives us another well-typed term of type α .

What this says is that *the types play no role in the actual computation*: typing constrains the terms we can write down in the first place, but once we've got a well-typed term, we just evaluate it exactly as we would an untyped term.

Fixed point combinators?

Can we encode recursion with fixed point combinators in the typed λ -calculus?

Let's try to give a type to the Curry fixed-point combinator

$$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)).$$

Obviously this means we need to type the subterm xx .

What type could it have?

Self-application?

Suppose the term xx has some type α .

That would mean that x is a function which returns things of type α , i.e. x has type $\beta \rightarrow \alpha$ for some β .

We want to apply this function to x , so x has to have type β as well. Since terms cannot have more than one type, we need

$$\beta \equiv \beta \rightarrow \alpha.$$

But our types are all finite things, so there is no such β .

So Curry's fixed point combinator cannot be typed.

No fixed point combinator

There is in fact *no* way of writing a fixed point combinator in the simply typed λ -calculus.

Indeed, the calculus is not Turing-complete, and in fact it is *strongly normalizing*:

Strong normalization

In the typed λ -calculus, every sequence of β -reductions eventually reaches a normal form.

Strong normalization

Strong normalization is a very strong property: it says that no matter how you reduce a term, you will eventually reach a normal form.

The Church-Rosser property and Church-Rosser theorem carry straight over from the untyped case, so this means

every term of the typed λ -calculus has a unique normal form, and employing any reduction strategy will eventually lead you to it.

It means that we can compute β -equality of terms, just by evaluating terms to normal form and seeing if they match.

It also means that there must be computable functions we cannot encode.

Programming in the typed λ -calculus

Let's see if we can encode pairs, numbers and so on in the typed λ -calculus as we did in the untyped case.

First let's look at pairing. Suppose we want to pair a term M of type α with a term N of type β . Then we'd use a term like

$$\lambda s. s M N.$$

What type should s have?

We apply it to M , and then apply that to N , so s must have a type like $\alpha \rightarrow \beta \rightarrow \gamma$.

We'd later like to use s as a selector to get back M and N . But that would mean s would return something of type α one time, and β another. We can't do that.

Our encoding of pairing fails. Oh no!

Numerals?

Since we can't create pairs, we can't use the Barendregt encoding of numbers either.

The Church numerals can be encoded, though. Remember that a Church numeral is a term like

$$\lambda f. \lambda x. f(f(f(x))).$$

If x has type α , then f must have type $\alpha \rightarrow \beta$ for some β , so that $f(x)$ makes sense.

But then $f(x)$ has type β , and we want to form $f(f(x))$, so we must have $\alpha \equiv \beta$.

It turns out that, for any α , any Church numeral

$$\lambda f^{\alpha \rightarrow \alpha}. \lambda x^{\alpha}. f(f(\dots(f(x))))$$

is a well-typed term of type $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$.

Booleans and conditional?

In the untyped case, we used pairing to set up our encoding of Booleans (truth values) and the conditional.

In the typed case, we don't have pairing, but we can still perform the encoding of Booleans: notice that pairing only failed when we tried to pair up terms of different types.

We can set

$$\begin{aligned}T &= \lambda x^\alpha. \lambda y^\alpha. x \\F &= \lambda x^\alpha. \lambda y^\alpha. y \\C &= \lambda b^{\alpha \rightarrow \alpha \rightarrow \alpha}. \lambda m^\alpha. \lambda n^\alpha. b m n\end{aligned}$$

to encode conditionals, but it is a little unsatisfactory: we need truth values at each type!

That is, the truth values we use for choosing between two terms of type α are different from the ones we use for type β .

Applied λ -calculi

In the field of typed λ -calculi, rather than trying to encode everything, it is much more common to add some *types* and *constants* to the language.

For instance, we might decide that there should be a special type `bool` for truth values, and add constants:

- ▶ `true` and `false` of type `bool`
- ▶ C_α of type `bool` $\rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$ for each type α

to our collection of well-typed terms.

We can then add so-called *delta rules* so that our extended language can be reduced, e.g.

$$\begin{aligned}C_\alpha \text{ true } M N &\rightarrow_\delta M \\C_\alpha \text{ false } M N &\rightarrow_\delta N\end{aligned}$$

Careful!!

When we add new reductions to the language, we might break some of the nice properties, such as the Church-Rosser property.

It turns out that, in the *untyped* λ -calculus, if we add constants `cons`, `car` and `cdr` and rules

$$\begin{aligned}\text{car}(\text{cons } M N) &\rightarrow_\delta M \\ \text{cdr}(\text{cons } M N) &\rightarrow_\delta N \\ \text{cons } (\text{car } M) (\text{cdr } M) &\rightarrow_\delta M\end{aligned}$$

then we lose the Church-Rosser property.

In a typed version, where we also add a new kind of type $\alpha \times \beta$ for pairs, we don't lose the CR property. Weird.

Constants for programming

Adding the right types, constants and rules to the typed λ -calculus turns it into something that looks a lot more like a “real” programming language.

We can restore the power of recursion to the language in lots of ways, the easiest of which is just to add Y combinators: for every type α , add a constant Y_α with type $(\alpha \rightarrow \alpha) \rightarrow \alpha$, and a rule

$$Y M \rightarrow M(Y M).$$

Similarly we can add numbers, arithmetic operators and so on.

Gödel's System T

Gödel introduced a language that we can think of as an applied typed λ -calculus with types and constants for truth values and natural numbers, plus a collection of *recursion* operators.

For every type α there's a constant R_α of type

$$\alpha \rightarrow (\alpha \rightarrow \mathbf{nat} \rightarrow \alpha) \rightarrow \mathbf{nat} \rightarrow \alpha$$

with rules

$$\begin{aligned} R M N 0 &\rightarrow_\delta M \\ R M N S(n) &\rightarrow_\delta N (R M N n) n \end{aligned}$$

Here S is the successor function on the natural numbers.

Gödel's System T

It turns out that System T is still *strongly normalizing*, but it can compute every function which is *provably total*.

That is, if one can prove in a formal logical system that a function on the natural numbers always gives an answer, then System T can encode that function.

We're now getting into the beginnings of the interesting parts of computability theory and mathematical logic. Time to stop!